Multimedia Tools and Applications
https://doi.org/10.1007/s11042-018-6940-2

MioStream: a peer-to-peer distributed live media streaming on the edge

Servio Palacios[1] · **Victor Santos**[1] · **Edgardo Barsallo**[1] · **Bharat Bhargava**[1]

Received: 3 July 2018 / Revised: 1 November 2018 / Accepted: 23 November 2018 /
Published online: 03 January 2019

Abstract

The typical centralized cloud model is poorly suited to latency-sensitive applications requiring low-latency and high-throughput. This paper proposes an integrity-preserving serverless framework for live-video streaming that runs on the edge of the network. We present the design, implementation, and evaluation of a novel P2P service based on WebRTC (web browsers with Real-Time Communications) called MioStream. MioStream is an open-source alternative for distributed media streaming that runs on the edge of the network without incurring in costly and extensive CDN infrastructure. We contribute a unique mix of algorithms using WebRTC data channels. For instance, under network degradation and high-churn environments, MioStream restructures the topology dynamically. MioStream provides authentication, privacy, and integrity of video chunks. This paper exposes a set of micro-benchmarks to measure the quality of service under network degradation and high churn environment (inducing failures). The Mesh topology offers the highest goodput per peer; the stalled playback on a node equals 1.8% of the total video play. Our results show the feasibility of this proof of concept under high-churn environments. The total stream interruptions in the topology are not longer than one second under a binomial distributed series of failures. The integrity check applied to each package includes a considerable overhead and impact the quality of service.

Keywords Edge computing · Event-driven model · Media streaming · Peer-to-Peer · WebRTC

✉ Servio Palacios
spalacio@purdue.edu

Victor Santos
vsantosu@purdue.edu

Edgardo Barsallo
ebarsall@purdue.edu

Bharat Bhargava
bbshail@purdue.edu

[1] Computer Science Department, Purdue University, West Lafayette, IN 47907, USA

1 Introduction

Peer-to-Peer (P2P) systems constitute the backbone of a diversity of distributed implementations (e.g., video streaming, file sharing, etc.) due to their resiliency and scalability. In particular, we are interested in an emerging technology called WebRTC that allows Real-Time Communication (RTC) on top of the World Wide Web (i.e., through web browsers such as Chrome, Firefox, Opera, and Edge). In their paper, Rhinow et al. provided an analysis of the feasibility of implementing live video streaming into web applications [22]. Although they offered performance measures, they did not include an open source repository, nor they studied the security implications and quality of service under failures. Lopez et al. introduced Kurento Media Server an open source WebRTC media server that offers features such as group communication, recording, routing, transcoding, and mixing [13]. They provide APIs that facilitates the development of web-based video applications. Lopez et al. did not present an analysis of the quality of service under failures or security checks (i.e., adversarial environment). Garcia et al. introduced NUBOMEDIA an open source cloud platform as a service (PaaS) designed for WebRTC services [5]. NUBOMEDIA exposes a set of APIs that facilitate the development of WebRTC applications.

In this paper, we propose *MioStream* a new distributed peer-to-peer video streaming system. MioStream takes advantage of emerging technologies to accomplish the desired operability. MioStream offers media streaming capabilities on top of WebSockets, WebRTC, and JavaScript. We identified challenges analogous to the ones exposed in [8, 9]. For instance, how to deal with peer selection in a high churn and heterogeneous network. MioStream offers authentication and integrity capabilities. For instance, We authenticate every peer against a centralized server, supervisor, using an authentication protocol based on TLS and DTLS cookies. Once authenticated, peers are monitored including their connection channels using WebSockets heartbeats and WebRTC data channel features. In the case of network degradation, the system is capable of restructuring the topology dynamically (i.e., changing active receivers or bridges) to obtain the performance in the overall network similar to [8, 9]. MioStream is an open-source alternative for distributed media streaming that runs on the edge of the network without incurring in costly and extensive CDN infrastructure.

In summary, our contributions are:

- A personalized object-lookup utilizing WebSockets and WebRTC.
- A novel communication and security layer implemented through the *supervisor* as Certification Authority (CA).
- An open source implementation and evaluation of the proposed techniques [16].
- An analysis of the quality of service in the presence of failures and attacks.
- We enhanced previous work on integrity validation of video chunks [7]. We contribute a unique method applied to live-video streaming.

Through a set of micro benchmarks, our evaluation demonstrates MioStream high-quality video streaming in the presence of failures.

A brief explanation of the building blocks of the system is in Section 2. The key concepts and architecture of the system are in Section 3. We provide an extensive set of experiments demonstrating the feasibility of our system in Section 4. Section 5 provides an insight into challenges experienced throughout the development of the system and future work. We include a discussion of related research and existing solutions (Section 6). Finally, we present our conclusions in Section 7.

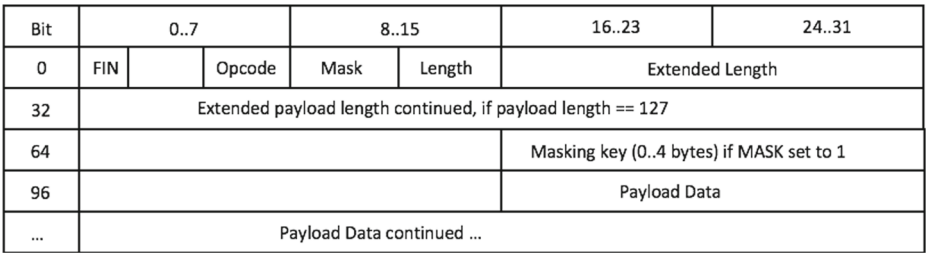

Fig. 1 WebSocket frame [6, 30]

2 Background

In this section, we introduce the essential concepts of the WebRTC stack. Our discussion is based on [6, 14, 35–37].[1]

2.1 WebSockets

The RFC6455 [30] describe the WebSockets as a communication protocol that provides full-duplex communication channels. Also, WebSockets provide message-oriented streaming of binary and text data [6]. According to [6], the API exposes the following features:

- Connection negotiation and same-origin policy enforcement.
- Interoperability with existing HTTP infrastructure.
- Message-oriented communication and efficient message framing.
- Sub-protocol negotiation and extensibility.

MioStream uses SocketCluster framework [27] to provide WebSocket services. SocketCluster supports both direct client-server communication (similar to *socket.io*) and group communication via *pub/sub* channels. Also, this library implements features such as heartbeats, timeouts, support for automatic reconnects, and multi-transport fall-back functionality as recommended in [6, 14]. SocketCluster is designed to scale both vertically across multiple CPU cores and horizontally across multiple machines/instances.

2.1.1 WebSocket frame [6]

Figure 1 shows a map of a WebSocket frame for the RFC6455 [30]. The FIN bit denotes if the frame is the last/final fragment of a message. The opcode comprises four bits that indicate the type of frame: (1) text or (2) binary, (10) for connection liveness checks.

2.1.2 Deploying WebSocket infrastructure

Since we do not have control over the policy of the client C_i network, MioStream have to use TLS tunneling over a secure end-to-end connection. Therefore, WebSocket traffic can bypass all the intermediate proxies and firewalls. It is possible to send security parameters to *SocketCluster* to allow secure connections. The following discussion is motivated by [6, 10, 14].

[1] We refer the reader to this useful books. You will find a detailed explanation of WebSockets and WebRTC

2.1.3 Performance objectives

- MioStream leverages reliable deployments through secure WebSocket (WSS over TLS).
- MioStream optimizes the payloads to minimize transfer size.
- MioStream considers compressing UTF-8 content to minimize transfer size.
- MioStream avoids *head-of-line blocking* splitting long messages.
- MioStream monitors the amount of buffered data on the client.

2.1.4 Performance notes

MioStream takes into consideration the performance criteria based on the WebSocket architecture. For instance, in-order delivery of WebSockets messages related to the order of the client's queue. Queued messages or large messages will delay delivery of messages queued behind it (e.g., *head-of-line blocking*). The application layer can split large messages into smaller chunks or implement its personalized priority queue. As a result, considering that the application is delivering latency-sensitive data, MioStream takes care of the payload size of each message.

2.2 WebRTC

Web Real-Time Communication (WebRTC) is a combination of standards, protocols, and JavaScript APIs, which enables peer-to-peer audio, video, and data sharing among browsers (peers) [14, 33, 35, 36]. WebRTC transports its data through UDP because latency and timeliness are critical [14, 35].

2.2.1 WebRTC protocol stack components

Figure 2 shows the protocol stack.

- ICE: Interactive Connectivity Establishment [11]
- STUN: Session Traversal Utilities for NAT [26].
- TURN: Traversal Using Relays around NAT [32].
- SDP: Session Description Protocol [25].
- DTLS: Datagram Transport Layer Security [2]
- SCTP: Stream Control Transport Protocol [28]

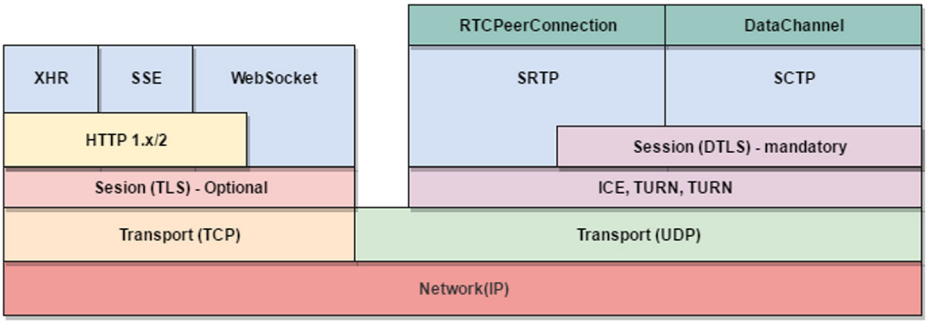

Fig. 2 WebRTC protocol stack [6, 35]

– SRTP: Secure Real-Time Transport Protocol [31]

To establish and maintain a peer-to-peer connection over UDP, ICE, STUN, and TURN are required. MioStream includes secure data transfers between peers through WebRTC; therefore, DTLS is used to encrypt data transfers [6, 14]. SCTP and SRTP expose multiplexing of different streams. Also, SCTP and SRTP provide partially reliable delivery on top of UDP and congestion control.

2.2.2 RTCPeerConnection API

RTCPeerConnection wraps in one interface the management, the connection setup, and state [6].

2.2.3 DataChannel

The peers can exchange arbitrary application data through the DataChannel API [6]. According to [6] each data channel can provide the following:

– Out-of-order or in-order delivery of messages.
– Partially reliable or reliable delivery of messages.

When setting a time limit or a maximum number of retransmissions the channel can be configured to be partially reliable [6]. Also, the WebRTC stack will take care of the acknowledgments and timeouts [6].

MioStream deals with the NAT traversal problem using WebRTC capabilities. Also, MioStream wrapped the PeerJS [20] library to provide WebRTC basic functionality and functions (e.g., Communication Layer). The built-in ICE [11] protocol performs the necessary routing and connectivity checks. MioStream's unique communication and security layer handle the delivery of notifications (*signaling*) and *initial session management*. MioStream integrates the PeerJS library with the communication and security layer.

2.2.4 Session description protocol [25]

WebRTC utilizes the Session Description Protocol (SDP) to define the parameters of the peer-to-peer connection. SDP describes the session profile; that is, SDP does not deliver media itself. Also, the Session Description Protocol describes the properties of a session through a simple text-based protocol. Finally, the JavaScript Session Establishment Protocol [12] (JSEP) encapsulates the Session Description Protocol (SDP); therefore WebRTC applications do not have to deal with SDP directly.

2.2.5 Interactive Connection Establishment (ICE) [11]

When establishing a peer-to-peer connection, the peers need to route packets to each other. Also, the peer-to-peer connection can fail because the peers can be on different private networks. To combat this, ICE agents manage this connectivity complexity. As shown in Fig. 2, each RTCPeerConnection object contains an ICE agent.

The ICE agent collect the candidates port tuples and local IP addresses. Also, the agent transmits connection keep-alive. Finally, the ICE agent makes connectivity checks among peers.

3 Design and implementation

3.1 Architecture

MioStream is composed of daemons installed with just one command via the NPM (Node Package Manager [18]). The system has a peer-to-peer architecture which comprises three main components, as shown in Fig. 3. We describe the components as follows.

- **Broadcaster**: The *broadcaster* is the source of a particular media stream. This is a multiplatform [3] application which leverages cutting edge web technologies such as WebRTC, Media Source Extensions [15], and all experimental components included in the Chromium [1] browser. With all Node.js native capabilities, the *Broadcaster* extracts binary data either from a camera/microphone hardware or a pre-encoded WebM [34] container, which can be audio or video. Finally, the *Broadcaster* streams the source media via WebRTC data channels (See Section 2) to the neighbors (e.g., one-hop apart connected peers).
- **Receiver**: The *receiver* is an *HTML5* web application which receives the desired media stream. Using several technologies present in the *Broadcaster* (WebRTC and Media Source Extensions), the *receiver* consumes a stream and **relays** the stream to its neighborhood (neighbors peers).
- **Supervisor**: The *supervisor* registers each peer (*broadcaster* or *receiver*). The *supervisor* monitors every connection to the components in real-time and notifies every relevant change to the corresponding nodes, i.e., *disconnection of a peer*. The supervisor includes the authentication protocol explained in Section 3.4. That is, we use the *supervisor* as a CA (Certification Authority) to take care of the authentication of every peer facing challenges similar to [17]. The system enhances the privacy and integrity of the video chunks allowing only authenticated peers to transmit/receive data and connect to streams. We assume that the peers have a valid public key that the server can validate in the authentication protocol. Also, the **authenticated** *broadcaster* generates keyed SHA-256 (HMAC) to ensure the integrity of video chunks. Therefore, the system allows the detection of malicious chunks in the broadcast.

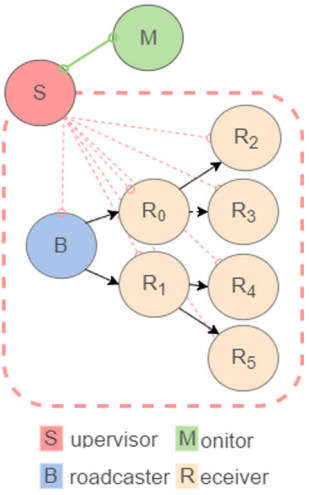

Fig. 3 MioStream architecture diagram

After a *Broadcaster* creates a stream, the *supervisor* registers and announces the existence of this new stream to the *Receivers*. When a *receiver* chooses to join the stream, the *supervisor* registers the new component in the stream topology. After the insertion of a new *receiver* in the topology, signaling events and connection setup connect this new peer to its neighbors directly. This new data channel creates a path where the stream will flow. Currently, we have implemented three topologies: **Linked List**, where viewers are chained one after another in the order of arrival. **Binary Tree**, where viewers connect into a tree structure. Also, we implemented a **Mesh** topology. The tree and mesh topologies have the advantage of the scalability and low content delivery latency. The stream only takes the *depth* of the tree to reach the last node in the topology, and the scalability of the overall topology follows the geometric series: $2^{n+1} - 1$ where n is the depth of the tree.

The following additional component, not part of the architecture, is used for administrative and experimental purposes.

– **Monitor**: The *monitor* is an *HTML5* web-tooling interface used to visually emit commands to the *receivers* and evaluate the effectiveness of a particular streaming topology. The monitor is conduct experiments on a streaming topology according to the following parameters: network topology for the receivers, number of induced failures per second, number of simulated attacks per second and probabilistic distribution to generate attacks or failures. Some of the features of the *monitor* include to measure stalled stream events, disconnections, apply different failure distributions (see Section 4), and visualize the shape of the topology (usually a binary tree).

The system relies on WebSockets and WebRTC data channels. We use the SocketCluster [27] library for the real-time signaling communication from which we leverage convenient features such as heartbeats, connection state, and event-driven communication. The *supervisor* is continuously listening for connections from *broadcasters*, and *receivers* (e.g., peers). At each disconnection, the supervisor updates the data structure to balance the load among peers. Also, the supervisor serves as a low-latency fault-tolerant messenger and certification authority.

3.2 Virtual topologies

One of the key aspects of MioStream is the virtual topologies. The main role of the virtual topology is to provide a deterministic way of arranging the connections between nodes (connected peer machines) in the system. When the supervisor component is deployed, it arranges the broadcasters and receivers in the desired topology data structure. In the case of a list, it uses a linked list to store each broadcaster following a receiver which in turn broadcast the packages to the next node. The flow of data in the topology is managed by monitoring delays and retransmissions in each section of the topology. The following is the description of each implemented topology and how they work:

1. **List Topology**: The simplest of the topologies is a linked list virtual overlay where each broadcasting node is connected to a receiver, which in turn is a broadcaster to the following node. Intuitively, the scaling of this topology is linear, and failures can be handled by creating connections to successive nodes. For instance, we can let specify that each node will connect at least to 3 nodes ahead of its current position, this way if a subsequent node fails, the stream can continue to flow uninterrupted or with little delay.

Stalled flows of data due to asymmetric connection speeds between nodes in the list can be mitigated by keeping the topology in descending order using connection speed and latency as the ordering metric.

2. **Mesh Topology**: The mesh topology is represented as a 2D matrix where each neighboring cell is a receiver node which consequently transmits the flow of data to neighboring nodes. This topology offers low latency and scalability by start the streaming the data from several broadcasters across the 2D matrix. These broadcasters are selected with the main criteria of creating a full coverage of topology at the moment of transmitting. For example, a broadcaster starts transmitting the package to all its 8 neighboring nodes(horizontal, vertical, and diagonal cells of the 2D matrix) and the data will start propagating like a wave caused by a water droplet. When a packet arrives at a node which has already received the data sequence, it drops the packet and doesn't propagate any further. Intuitively, we can guess that resilience is improved by how well the broadcaster are distributed across the topology and how well the packages propagate until they are dropped by overlapping 'waves'. Bottlenecks and asymmetric propagation speed is mitigated by arranging the broadcaster and neighboring nodes in a circular shape, i.e. the fastest node in the center(broadcaster) and slower nodes at the edges of the propagation wave. A drawback of this topology is the high complexity to maintain when nodes fail and the propagation coverage needs to be calculated and rearranged.
3. **Tree Topology**: The three topology is represented as a binary tree and has the best scalability of the three used topology in the study. Similar to the list topology, resilience can be increased by creating preventive connections to successive nodes. The dual connection of the binary tree makes it convenient to handle faults since connections can be created crossing the branches and follow protocols to reroute the traffic through those branches. Stalled flow can be mitigated to implement a max-heap instead of an unordered binary tree, this way preventing bottlenecks in slow link connected nodes.

These three topologies are a starting point to future and more complex structures such as a self-balancing directed graphs where the topology is ruled by a tradeoff of optimality in the data flow in exchange of maintainability in the presence of failures. Another possibility is to apply machine learning to weight the probability of failure in certain nodes by inspecting feature vectors of physical, software, and connection links features, this way optimizing the backup links among the topology.

3.3 The communication layer

We now explain the communication components that provide signaling features during a media streaming session. The communication layer abstracts all the burden of WebRTC, SocketCluster and the Security Library. This layer provides application transparency to the operations being executed under the hood.

1. *Collision Resistant Unique Identifier*: Since MioStream wraps the PeerJS library inside of the communication library, it needs to provide a collision resistant function to generate unique names with negligible probability of collisions. We use a randomly generated UUID which has 128 bits. The chance that 128 bits of having the same value can be calculated using probability theory (i.e., birthday problem), which can be shown to be negligible.

   $p(n) \approx 1 - e^{\frac{-n^2}{2*x}}$ where n is the number of bits.

   We use this *PeerID* to ensure unique WebRTC data channels between peers.

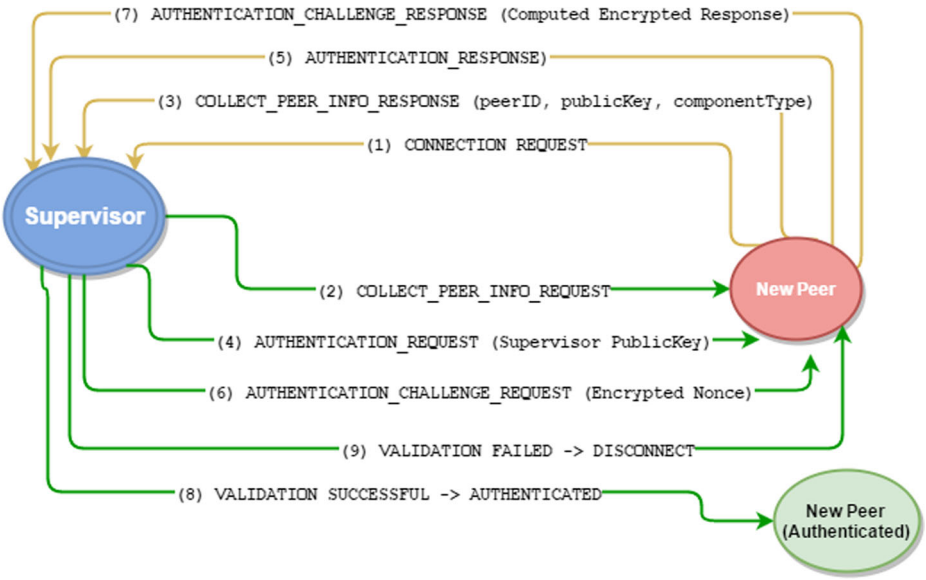

Fig. 4 Authentication finite state machine

2. *Public and private key*: Every peer has its own RSA private and public key.[2] We provide a parameter to configure the level of security enforcement among peers (e.g., key bit length $\geq$ 2048). These keys are used for the Authentication Protocol described in the Security Layer. Our security model assumes a trusted *supervisor*.
3. *Connected Peers*: Every peer keeps its neighborhood set similar to [4, 24]. We then forward packets received and/or send to the application layer, the *video chunks* received.
4. *Signaling Capabilities*: Every peer connects to the *supervisor* using the communication library. Consequently, this layer handles all the signaling, heartbeats, and disconnections.
5. *WebRTC Data-channels*: We want to be able to build a variety of topologies using our implementation. Hence, the communication library provides a capability that is the equivalent to creating an *edge* $e(u, v) \in E$ in a graph $G = (V, E)$ where u and v are authenticated nodes in our network.[3] Consequently, we can create a rich set of topologies using the basic building blocks of *edges* (e.g., data channels see §2) and *vertices* (e.g., peers, *receivers* or *broadcasters*).

3.4 Security layer

In this subsection, we discuss the security guarantees that our system offers.

1. *Authentication Protocol*: MioStream implements a handshake protocol, as shown in Fig. 4. Although we took some ideas from TLS and DTLS [17]; MioStream includes the *componentType* in the topology as a new property.

[2]Extending our library to include Elliptic Curve Cryptography is straightforward.
[3]In this paper, we use the terms nodes, vertices, and peers interchangeably.

- MioStream receives a socket connection to its SocketCluster server (*supervisor*).
- The *supervisor* sends the message `COLLECT_PEER_INFO_REQUEST`.
- The peer replies with a `COLLECT_PEER_INFO_RESPONSE`, including its PeerID (used in the PeerJS connection), the Public Key, and a $componentType$ (*Broadcaster* or *Receiver*).
- The *supervisor* sends its public key to the peer using `AUTHENTICATION_REQUEST` message.[4]
- The peer *ACKs* with the message `AUTHENTICATION_RESPONSE`.
- The *supervisor* now computes a nonce including two random numbers and encrypts those numbers using the peer's public key. We then send the nonce using `AUTHENTICATION_CHALLENGE_REQUEST`.
- The Peer receives the challenge request, decrypts the nonce, and computes a number based on two random numbers included in the nonce. The peer then proceeds to encrypt the response using the supervisor's public key. Finally, the peer answer with `AUTHENTICATION_CHALLENGE _RESPONSE`.
- The supervisor now checks the response given by the peers, if matches the one computed on the server, then the peer has now the state of **AUTHENTICATED** and is ready to initiate or receive streams.

2. *Integrity Validation of Video Chunks*: Habib et al. [7] proposed a probabilistic packet verification protocol. In particular, their *One Time Digest Protocol* (OTDP) utilizes a keyed hash to generate digests. Also, they divide the media file into segments or blocks; each block contains many packets. The server provides a secret key $K_i \in K$ for each segment i. Each supplying peer will generate digests using the provided keys and segments. Habib et al. motivate the integrity verification on video chunks presented in this project; therefore, to provide data integrity during peer-to-peer video streaming, MioStream adopts a similar approach but with some significant differences and contributions. First, this project only has a *broadcaster* for a specific stream. Second, we have implemented a personalized object (stream) lookup. Third, our implementation is live-oriented; hence we do not know all video-chunks in advance. The security model does not assume that connecting peers are trustworthy, so MioStream enforces authentication through the *supervisor* including a Public Key Infrastructure (PKI). Every *authenticated* peer is allowed to be part of our streaming sessions. The compute message digests are computed using a keyed SHA-256 (*HMAC*) and NodeJS Crypto library instead of digital signatures as in [7], but we use video chunks VC instead of segments. Therefore, we derived the following digest:

$$D_{ij} = h(K_i, VC_j)$$

where D_{ij} is the computed HMAC for the video chunk $(VC)\,j$ and K_i is the *supervisor's* generated key for stream i.

Our implementation modifies the steps of One Time Digest Protocol (OTDP) to adapt it to our model as follows:

- The peer P_0 authenticates itself against the *supervisor*.
- P_0 requests a list of media streams to the *supervisor*.
- The *supervisor* provides a set of keys based on the search results.

[4]Our future work will include a Certification Authority to deal with digital signatures, public keys, and tokens.

- When the *broadcaster* (P_i) starts the streaming, it sends an initialization packet to P_0.
- The *broadcaster* (P_i) sends both data and digests to P_0.
- P_0 verifies random video chunks. Furthermore, P_0 can verify every packet depending on the *threshold* defined as a parameter.

The *broadcaster* computes the hash of every video chunk and sends the packet via the data channels connected to its *receivers*. Next, the *receiver* checks for the integrity of the packet and compares with the digest sent by the *broadcaster* or source of the media stream. In contrast to [7], we do not know all chunks of media in advance. Therefore, MioStream cannot compute random message digests and send those to every peer in our topology. We compute message digests of the chunks generated by the application layer. The *receiver* can verify at random one message digest to identify forgeries. MioStream includes a probabilistic approach using a uniform random number generator and comparing those generated numbers against a *threshold* defined as a parameter. The size of the message digests are 44 bytes (Base64 encoded).

3.5 Discussion

The system has several limitations that are mainly due to client-server nature of web technologies. Currently, we can use two execution points: Any browser's ECMAScript run-time with WebRTC support, where users go to a website and automatically setup all the execution environment, and a Node.js application (*supervisor*) that currently runs on top of the V8 engine with I/O and OS's system call capabilities. The browser run-time is limited to its intrinsically high failure rate, especially in mobile devices. Although MioStream lacks fault tolerance capabilities; the implementation is flexible enough to include more SocketCluster instances that can provide redundancy among *supervisors* (i.e., the communication layer can be extended to accommodate fault tolerance features). MioStream offers authentication and data integrity protocols; however, security poses a significant challenge because of the usually heterogeneous and high-churn network. Still, we can alleviate the attack surface utilizing trusted hardware in the *supervisor* and *relay points*.

4 Experiments

The experiments evaluate two main aspects of the system: *video streaming quality* under failures and *integrity verification of video chunks* overhead. First, we injected failures into the network to simulate how users would abandon the video stream (i.e., we use three distributions Binomial, Uniform, and Poisson). Hence, P2P connections between nodes in the network topology are aborted and packets get lost, ultimately causing stalled playbacks on other peers. The quality of the services was evaluated in terms of stalled playback and goodput metrics, as we will refer later. Stalled playbacks refer to delays or pauses in the viewer. Goodput measures the number of bytes received by the application layer (video chunks). Topology simulates the connections among the peers (e.g., linked list, tree, mesh). Second, we evaluated the performance of our the integrity verification of video chunks implemented in the security layer.

4.1 Experiment setup

The experiment setup consisted of 83 machines with different specifications, ranging from four to eight 2.0 GHz cores, 8GB of RAM, connected to a 1Gbps local area network. All 83 machines act as *receivers*. In another machine, we set up the *supervisor* and the *monitor*. Finally, we stream the media from a separate machine running the *broadcaster*. Hence, we use a total of 85 machines for all experiments. In Table 1, we can see the technical specification of the video streamed for all experiments.

Each experiment is repeated on three different network topologies: *linked list*, *tree* and *mesh*. The *monitor* component has the capability of visualizing how nodes connect throughout the whole experiment. Basically, in the first set of experiments, we are particularly interested in observing the quality of service of the system under a non-reliable network (we use a pessimistic method of inducing failures).

4.2 Method of injecting failures

We define a failure as a disconnection of a *viewer* from the network. Since each *viewer* also forward the video to another peer, the flow of the stream is interrupted with every failure. When a *viewer* disconnects from the stream, that same node is reconnected to the network as a new incomer *viewer*. Hence, the network size is kept constant during the whole experiment. The rate of failures varies from one to five.

To better evaluate the behavior of the system in different failures scenario, we induce the failures based on three probability distribution: binomial, uniform, and Poisson. The *monitor* use the distribution to choose the failure peer from the network topology. For each distribution and network topology, we ran five experiments, varying the failure rate from one to five failures per second. The experiment consist on stream one minute of video across all the receivers. We took four samples in order to rule out spontaneous network congestion and processing load fluctuations.

Table 1 Media Information of the WebM video used for the experiment

Media Information		
General	Format	WebM
	Format version	Version 2
	File size	24.0 MiB
	Duration	2mn 53s
	Overall bit rate mode	Variable
	Overall bit rate	1159 Kbps
Video	Format	VP8
	Width	1920 pixels
	Height	1080 pixels
	Display aspect ratio	16:9
	Frame rate mode	Constant
	Frame rate	29.970
Audio	Format	Vorbis
	Bit rate	128 Kbps
	Channel(s)	2 channels
	Sampling rate	44.1 KHz

The metrics used to evaluate the behavior of the system were:

- *Stalled:* Overall stalled playbacks in the stream during the experiment.
- *Total Stalled Time:* Overall playback stalled time during the experiment.
- *Average Stalled Time:* Overall average stalled time during the experiment.
- *Goodput*. Average goodput, or bytes received per second, on each peer during the experiment.

These metrics all together will describe the quality of the stream on different failure rates and scenarios.

4.3 Results

Figures 5, 6, 7, 8, 9, and 10 show the results for the stalled counter and stalled time for each distribution and network topology, varying from one to five failures per second. Under the **uniform** distribution, every peer in the topology has the same probability to fail during the experiment. As we can see in the figures, the stalled counted and the total stalled time experiences a gradual increment as the failure rate increases.

The **binomial** distribution induces failures mostly at the center of the topology. The failure peers were selected using 20 trials with a 80% of success probability. As we can observe, in the Figures 5–10, both stalled counter and total stalled time are very low compared to those values in the uniform distribution experiments. The low stalled counter can be explained by analyzing the following scenario: A broadcaster sends a chunk of the media through all the topology at time t_1, and let say that this media chunk reaches the leaf nodes in the topology after n milliseconds of latency. If the binomial failure experiment induces failures at the center of the topology, it may be common that the chunks traveling the topology are in transit before the center, or after the center at the moment of failure. Since the chunk, if already delivered to the leaves or are before the center, there won't be stalled playbacks.

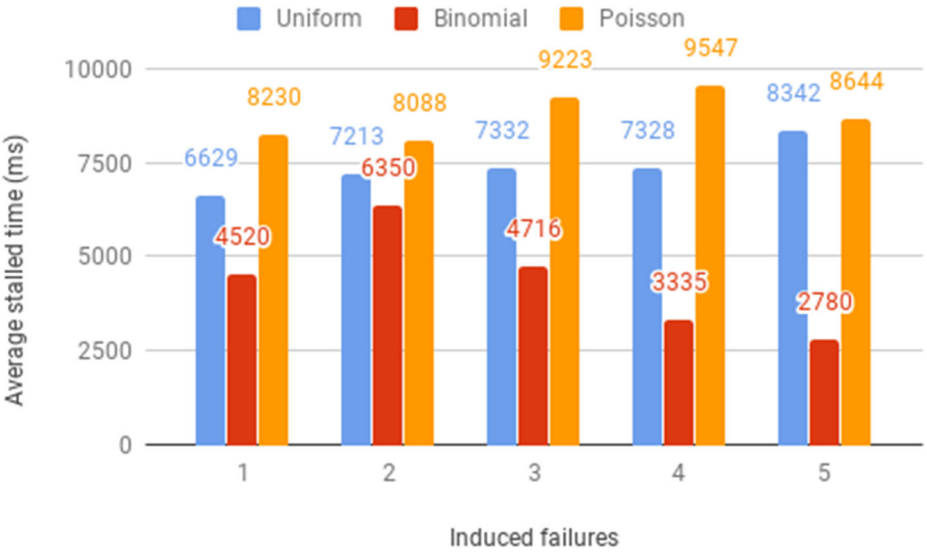

Fig. 5 Linked list topology under induced failures (Uniform, Binomial, and Poisson distribution). Average stalled time of playback per peer

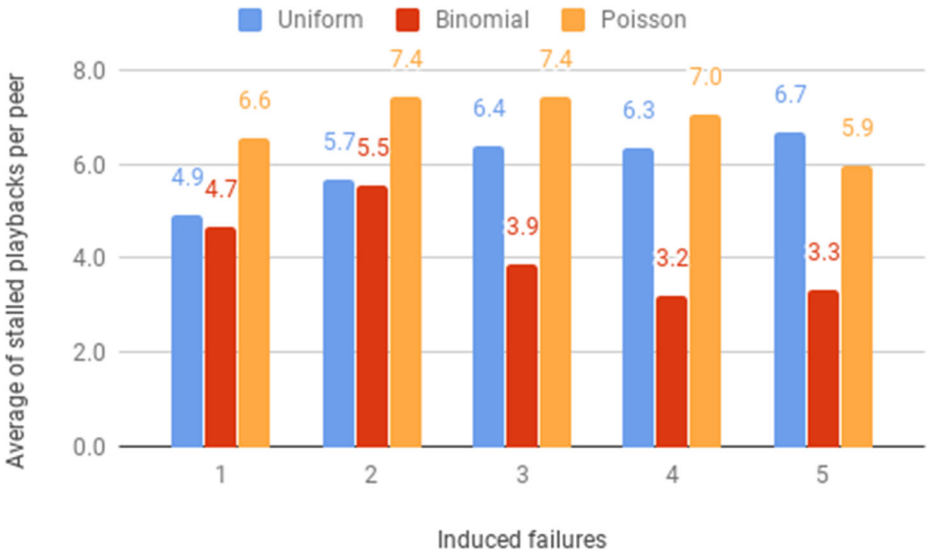

Fig. 6 Linked list topology under induced failures (Uniform, Binomial, and Poisson distribution). Average stalled counter per peer

Finally, with a **Poisson** distribution the stalled counter and stalled time also gradually increments with the number of failures. In the sense of the location of the induced failure in the topology, this distribution behaves similarly to the binomial but closer to the root or

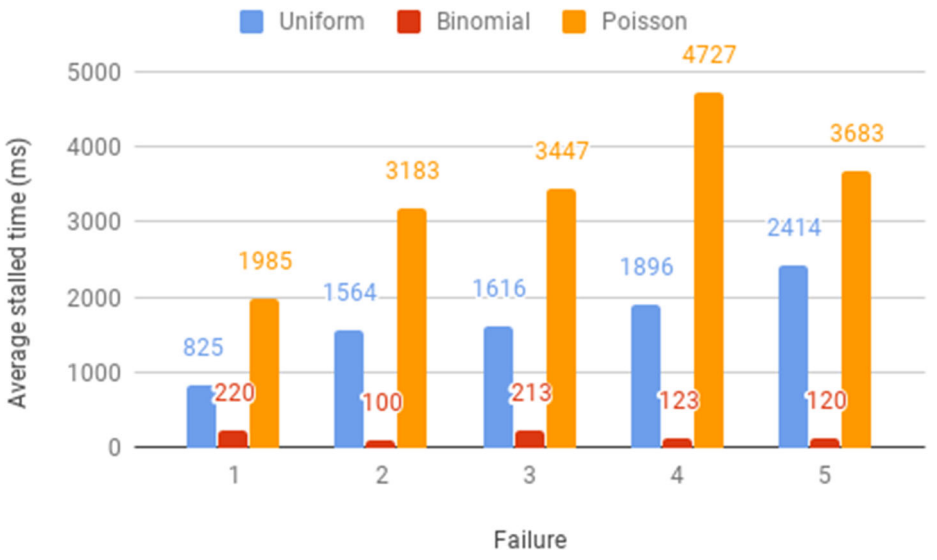

Fig. 7 Tree topology under induced failures (Uniform, Binomial, and Poisson distribution). Average stalled time of playback per peer

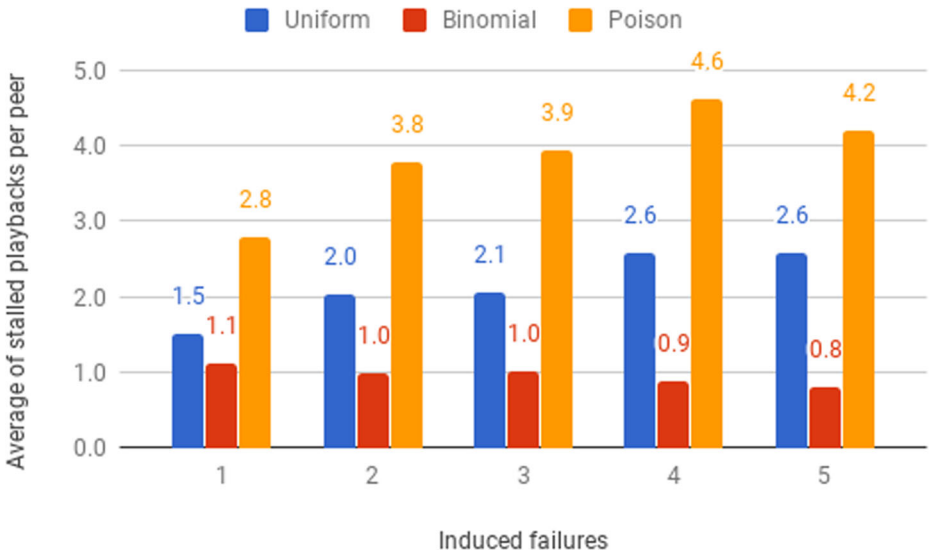

Fig. 8 Tree topology under induced failures (Uniform, Binomial, and Poisson distribution). Average stalled counter per peer

broadcaster instead of the middle of the topology. Therefore, the values are higher than the other two distributions, since failures closer to the broadcast (root) impact more peers in the topology. Each faulty peer was chosen with a mean of 83/2 (half number of nodes in the topology).

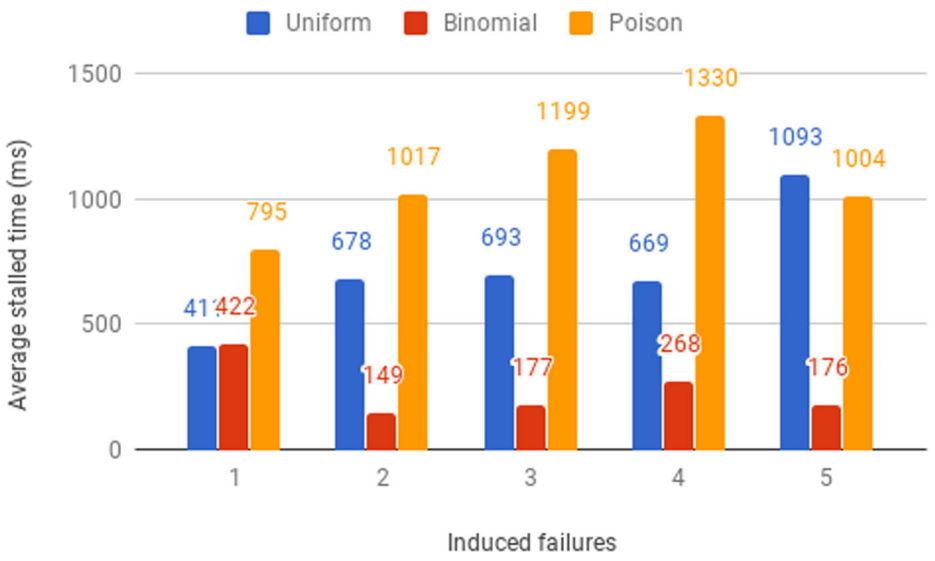

Fig. 9 Mesh topology under induced failures (Uniform, Binomial, and Poisson distribution). Average stalled time of playback per peer

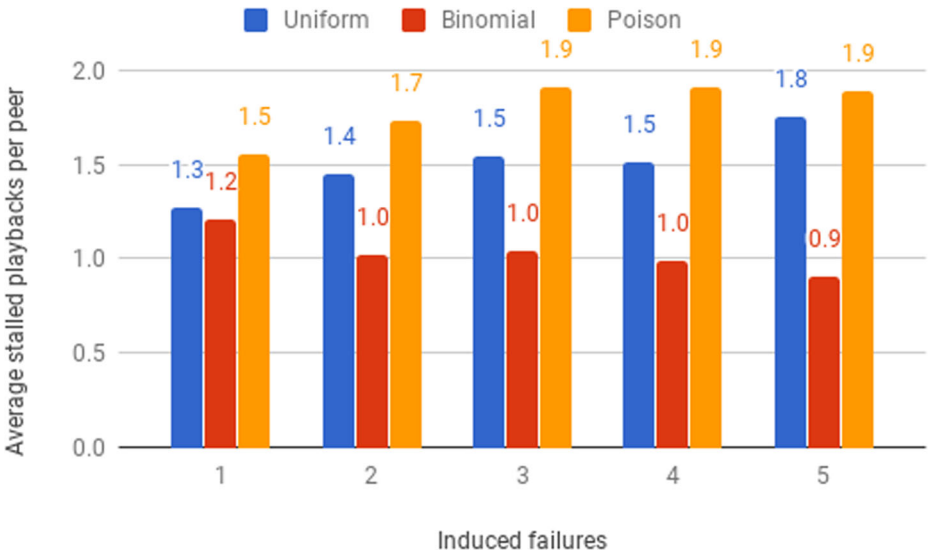

Fig. 10 Mesh topology under induced failures (Uniform, Binomial, and Poisson distribution). Average stalled counter per peer

4.4 Scalability and goodput

The Fig. 11 compares the resilience of each topology varying the failure rate. For the analysis, we choose the uniform distribution to have an equal probability of failure for each node of the topology. The viewers in the linked list topology experience the worst quality

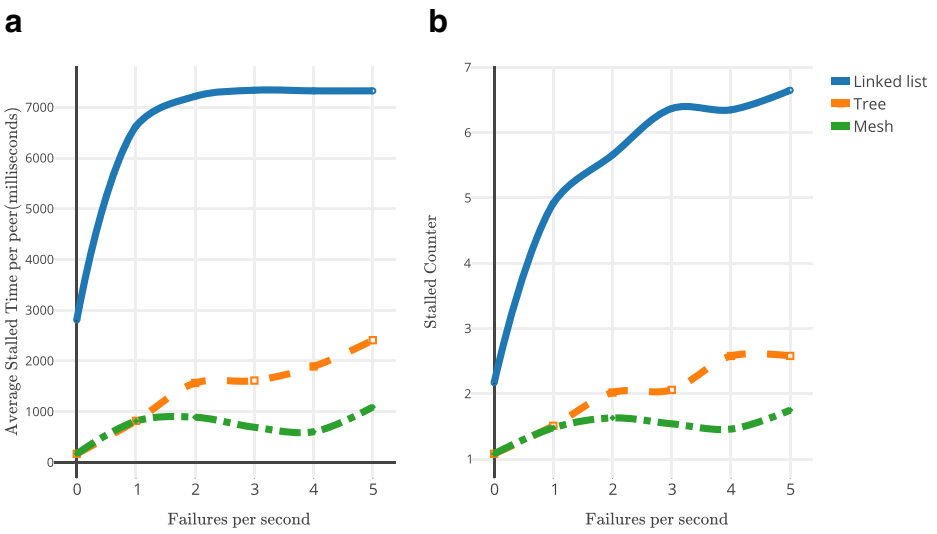

Fig. 11 Scalability under induced failures (Uniform, Binomial, and Poisson distribution) according to network topology. **a** average stalled time of playback per peer; **b** average stalled counter per peer

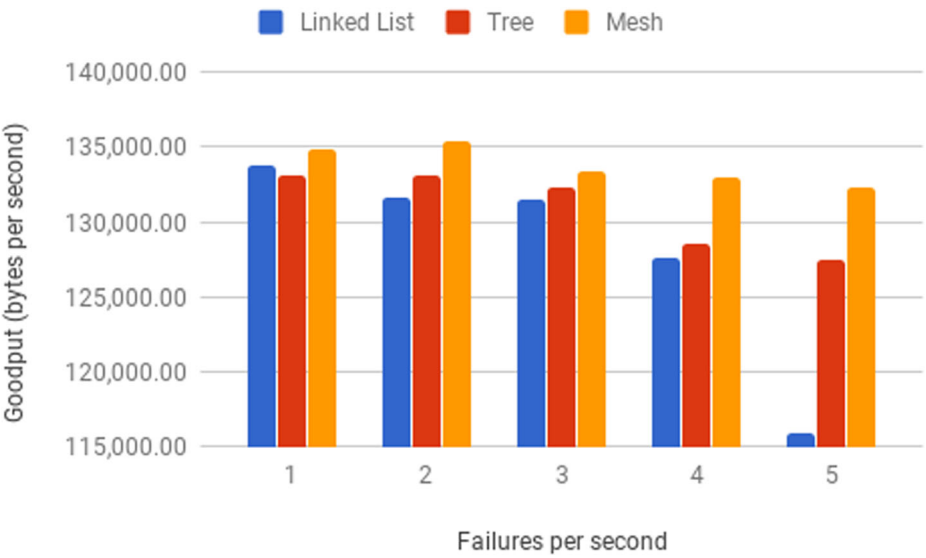

Fig. 12 Goodput (bytes per second) under induced failures using the uniform distribution

of service among the three topologies. A failure on this topology has the more significant impact since it affects the higher number of peers (recall that each peer is serially connected with each other). For instance, a failure on a peer close to the source (*broadcaster*) requires reconfiguring most of the network. The stalled count and the stalled time for the linked list topology at least double the values of the other two topologies. The stalled time per peer during the whole experiment goes from 2807 ms to 7325 ms for zero and five induced failures respectively.

The service quality increases drastically with more scalable topology like a tree. This topology is more resilient to failures and offers a shorter reconfiguration time in the presence of disconnection. However, the topology is highly vulnerable to failures near the root or source (*broadcaster*), as we observed earlier. The stalled time in this topology varies from 169 ms (an order of magnitude less than the linked list topology) to 2410 ms. However, the best quality of service is offered by the mesh topology. Even though the tree and mesh topology behave similar with minimal disconnections, the mesh topology is more robust to failures. The mesh offers faster reconfiguration and overcomes the vulnerability to failures near the source by providing more alternative connection than the tree topology. The peers in the mesh topologies experience fewer stalls with respect the other two topologies, and the average stalled time is less than half of the respective value in a tree. The stalled time goes from 169 ms (similar than tree topology) to 1093 ms. Furthermore, as we can observe from Fig. 12, the mesh network offers the highest goodput per peer, which is 10% than the goodput on the tree topology.

4.5 Integrity validation of video chunks overhead

We now discuss the total number of stalled video chunks due to the integrity validation. In Figs. 13 and 14 we can observe different scenarios using thresholds[5] starting from

[5]The threshold represents the probability of the integrity analysis of a video chunk.

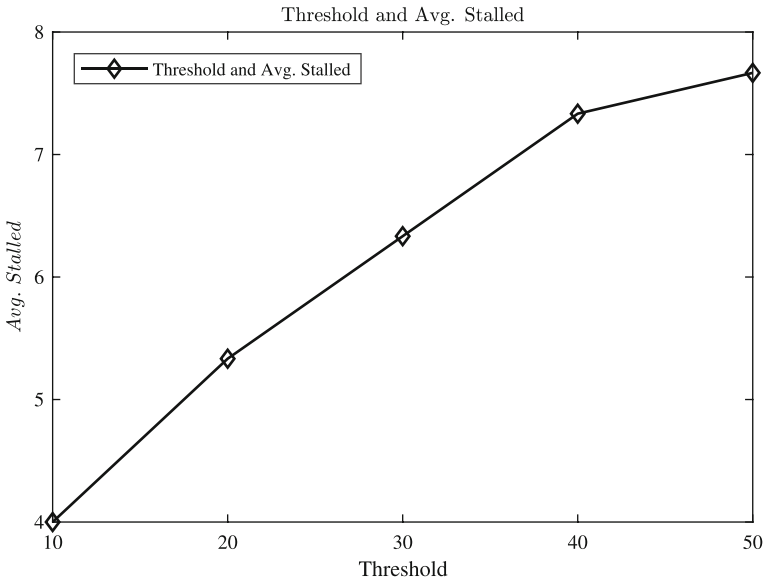

Fig. 13 Average number of stalled video chunks per threshold

10% up to 50%. The average delayed video chunks increase according to the threshold when the probability limit was below 10 percent the quality of service improved considerably.

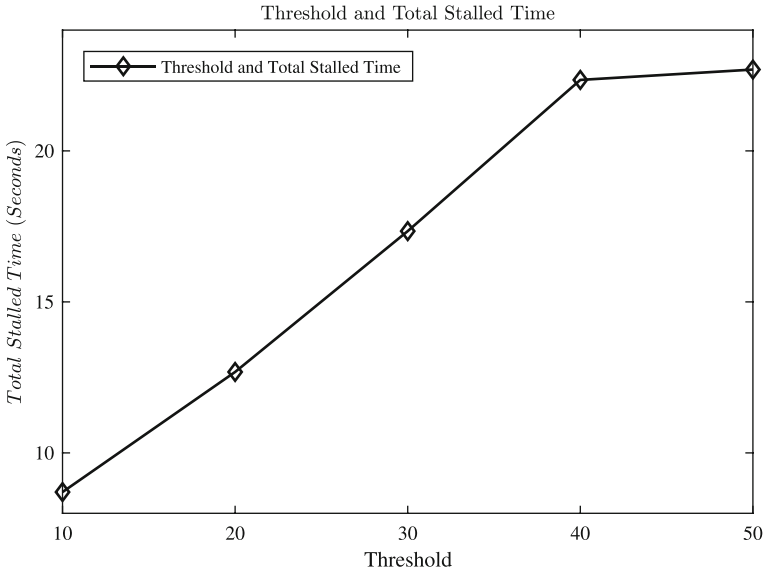

Fig. 14 Total stalled time per threshold

4.6 Security layer overhead

4.6.1 Experimental setup

Single-machine experiments were run using a machine with Intel Core i7 (4 cores @2.8GHz, 8MiB cache) with 16GiB of RAM. We tested the NodeJS Crypto Library [19].

4.6.2 HMAC

The experiment consists of 10 runs of HMAC instantiations and 10 HMAC digests generation. The Fig. 15 shows the overhead caused by this library per HMAC digest generation.

4.7 Analysis

The main contribution of our work is to expose a proof of concept of a decentralized (in the sense of content distribution) of a media content streaming using peer-to-peer communication. Also, we demonstrated that it is feasible to maintain considerably good quality in exchange for resource cost and network overhead. All experiments with different distributions and failure rates aimed to simulate real-life failure scenarios, and how the system will behave in such extreme conditions. So far, we have shown in Figs. 5–10, how the streaming quality is affected and how stalled playbacks scale as failure rate increases. Furthermore, the average stalled time in the overall topology is below one second in most scenarios, independent of the distribution. Hence, the total stream interruptions in the topology are not longer than one second, and in most cases are milliseconds which can be tolerated and perhaps be so brief that are not even perceived by the viewer.

Besides, the study offers a comparison between different network topologies, highlighting the benefits and drawbacks of each one. As shown in Figs. 11 and 12, the mesh topology is more robust to failures than the other topologies analyzed, offering a higher goodput and fewer stalled playbacks. Moreover, the stalled playback on a node equals 1.8% of the total video play. The amount is insignificant if we take into consideration that the experiments aimed to simulate extreme conditions, with as many as five failures per seconds. Finally,

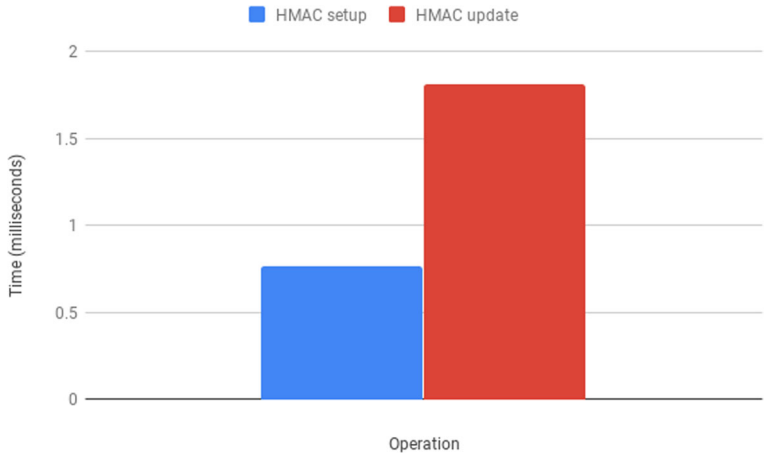

Fig. 15 HMAC setup and digest generation

the integrity check includes a considerable overhead; hence an acceptable threshold will be lower than ten percent of the total of chunks.

4.8 Use cases

We have shown how to maintain considerable good quality in a live stream topology using mainly three components: *supervisor*, *broadcaster*, *receivers*. We are presenting our system in the context of live media streaming. On the other hand, there is no reason why this same concept cannot apply to other scenarios which also relies on CDN to distribute content. Some of the potential scenarios to apply the same architecture are:

- *Mobile applications content distribution*: Recently, many mobile applications (i.e., social media), distribute media content in a graph fashion. I.e., friends to friends. This case will be even simpler than our scenario since the stream of content is not real-time. Implementing a similar architecture would significantly reduce the cost of scale-out large mobile applications which need to distribute media content.
- *Software Updates*: Modern software, such as those built with Electron [3], supports hot updates, meaning that the application can be downloaded in the background and installed inside of a sandbox without the need of natively uninstall and install a different binary. The same concept applies to distributed hot updates without requiring large content distribution networks.

5 Future work

In the experiments section, we showed the potential of our system. However, the system should have many other qualities to be considered robust. We aim to improve the quality of the streaming further and prevent stalled playbacks. First, MioStream will include high-availability in case of failures through *supervisor* replication in our topology. Second, MioStream's communication layer will detect *relay points* through machine learning techniques. The relay points aim to avoid stalled playbacks and congested paths. Finally, MioStream will provide new enhancements in the security layer (e.g., Elliptic Curve Cryptography).

6 Related work

MioStream shares some characteristics as Peer2View [23]. For example, Peer2-View is built on top of UDP-based transport library which provides reliability and security guarantees. Also, they authenticate their users against a central authority (e.g., in our case the *supervisor*). They provide encryption using SSL and integrity. However, there are many differences between MioStream and Peer2View; the latter solves the NAT problem with NATCracker [23], the former uses WebRTC, ICE, STUN and TURN. Moreover, Peer2View is a commercial peer-to-peer live video streaming (P2PLS) using Content Distribution Network (CDN). In contrast, MioStream is an Open Source system.

We now compare against an Energy-Efficient Mobile P2P Video Streaming [38]. Their goal is to minimize and balance the energy consumption of participating devices in the video streaming session. Similarly, our goal is to provide the best quality of video streaming across all participants of our video streaming session balancing the traffic in the network using data structures to provide congestion control (maximizing network utilization). Wichtlhuber et al. have energy consumption as its primary goal; we can extend our vision and minimize

the usage of mobile devices to avoid degradation of our network and maximize battery life in mobile devices.

We based much of our work on PROMISE and CollectCast [7–9]. However, there are many important differences. For instance, we assume constant bandwidth for each peer (e.g., we do not have the notion of partial bandwidth contribution, MioStream uses all bandwidth available on the WebRTC data channel), and there is only one *broadcaster* for every stream. Available streams are provided by the *supervisor* to the *receivers*. In [8, 9] they exploit the properties of the underlying network in which one receiver can collect data from multiple senders. PROMISE is independent of the underlying P2P network. Therefore, PROMISE can be deployed using Pastry [24], Chord [29], and CAN [21]. MioStream depends on WebSockets and WebRTC architecture; the *supervisor* exposes the object lookup. Furthermore, MioStream implements its personalized object lookup, called *Stream Manager*.

In their paper, Rhinow et al. provided an analysis of the feasibility of implementing live video streaming into web applications [22]. Although they offered performance measures, they did not include an open source repository, nor they studied the security implications and quality of service under failures. MioStream exposes an analysis of QoS in the presence of failures. Lopez et al. introduced Kurento Media Server an open source WebRTC media server that offers features such as group communication, recording, routing, transcoding, and mixing [13]. They provide APIs that facilitates the development of web-based video applications. Lopez et al. did not present an analysis of the quality of service under failures or security attacks. Garcia et al. introduced NUBOMEDIA an open source cloud platform as a service (PaaS) designed for WebRTC services [5]. NUBOMEDIA exposes a set of APIs that facilitate the development of WebRTC applications.

7 Conclusion

An increasing number of companies require serverless privacy-preserving frameworks for live-video streaming. We presented the design, implementation, and evaluation of a novel P2P service based on WebRTC (web browsers with Real-Time Communications) called MioStream. MioStream directly targets this essential and practical application. This paper contributes an open-source alternative for distributed media streaming that runs on the edge of the network without incurring in costly and extensive CDN infrastructure. We contributed a unique mix of algorithms using WebRTC data channels. Also, under network degradation and high-churn environments, MioStream restructures the topology dynamically. MioStream provides authentication, privacy, and integrity of video chunks.

This paper exposed a set of micro-benchmarks to measure the quality of service under network degradation and high churn environment (inducing failures). Failures are induced in the topology using three distributions (e.g., Uniform, Binomial, and Poisson) and three network topologies such as mesh, tree, and linked list. The Mesh topology offers the highest goodput per peer; the stalled time goes from 169 ms to 1093 ms, the stalled playback on a node equals 1.8% of the total video play. Our results show the feasibility of this proof of concept under high-churn environments. We conclude that the total stream interruptions in the topology are not longer than one second under the binomial distribution, and in most cases are milliseconds. The integrity check includes a considerable overhead.

References

1. Chromium, https://www.chromium.org
2. Datagram Transport Layer Security (DTLS), https://tools.ietf.org/html/rfc4347
3. Electron, http://electron.atom.io
4. Ferreira R, Jagannathan S, Grama A (2006) Locality in structured peer-to-peer networks. J Parallel Distrib Comput 66(2):257–273. https://doi.org/10.1016/j.jpdc.2005.09.002
5. Garcia B, Lopez L, Gortzar F, Gallego M, Carella GA (2017) NUBOMEDIA: The first open source WebRTC PaaS. In: Proceedings of the 2017 ACM multimedia conference, pp 1205–1208. https://doi.org/10.1145/3123266.3129392
6. Grigorik I (2013) High performance browser networking. O'Reilly Media, Sebastopol
7. Habib A, Xu D, Atallah M, Bhargava B (2005) Verifying data integrity in peer-to-peer video streaming. CERIAS. https://doi.org/10.1117/12.587201
8. Hefeeda M, Habib A, Botev B, Xu D, Bhargava B (2003) PROMISE: peer-to-peer media streaming using collectcast. In: Proceedings of the 2003 ACM multimedia conference. https://doi.org/10.1145/957013.957022
9. Hefeeda M, Habib A, Botev B, Xu D, Bhargava B (2005) CollectCast: a peer-to-peer service for media streaming. In: Proceedings of the 11th ACM international conference on multimedia, vol 11, no 1, pp 68–81. https://doi.org/10.1007/s00530-005-0191-6
10. How does HTTP/2 solve the Head of Line blocking (HOL) issue, https://community.akamai.com/customers/s/article/How-does-HTTP-2-solve-the-Head-of-Line-blocking-HOL-issue
11. ICE, https://tools.ietf.org/id/draft-ietf-ice-rfc5245bis-13.html
12. JavaScript Session Establishment Protocol (JSEP), https://tools.ietf.org/html/draft-ietf-rtcweb-jsep-24
13. Lopez L, Paris M, Carot S, Garcia B, Gallego M, Gortazar F, Benitez R, Santos J, Fernandez D, Vlad RT, Gracia I, Lopez FJ (2016) Kurento: The WebRTC modular media server. In: Proceedings of the 2016 ACM multimedia conference, pp 1187–1191. https://doi.org/10.1145/2964284.2973798
14. Loreto S, Pietro RS (2014) Real-time communication with webRTC: Peer-to-peer in the browser. O'Reilly Media, Sebastopol
15. Media Source Extensions, https://w3c.github.io/media-source
16. MioStream, https://github.com/maverick-zhn/miostream
17. Modadugu N, Rescorla E (2004) The design and implementation of datagram TLS. In: Proceedings of The 2004 Network and Distributed System Security Symposium (NDSS). https://doi.org/10.1.1.74.6613
18. Node Package Manager (NPM), https://www.npmjs.com/
19. NodeJS Crypto Library, https://nodejs.org/api/crypto.html
20. PeerJS, http://peerjs.com/
21. Ratnasamy S, Francis P, Handley M, Karp R, Shenker S (2001) A scalable content-addressable network. In: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications. https://doi.org/10.1145/964723.383072
22. Rhinow F, Veloso PP, Puyelo C, Barrett S, Nuallain EO (2014) P2P Live Video Streaming in WebRTC. In: World Congress on Computer Applications and Information Systems (WCCAIS). https://doi.org/10.1109/WCCAIS.2014.6916588
23. Roverso R, El-Ansary S, Haridi S (2012) Peer2View: A peer-to-peer HTTP-live streaming platform. In: IEEE 12th international conference on peer-to-peer computing. https://doi.org/10.1109/P2P.2012.6335813
24. Rowstron A, Druschel P (2001) Pastry: scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In: Proceedings of the 18th IFIP/ACM international conference on distributed systems platforms (Middleware), pp 329–350. https://doi.org/10.1007/3-540-45518-3
25. Session Description Protocol (SDP), https://tools.ietf.org/html/rfc4566
26. Session Traversal Utilities for NAT (STUN), https://tools.ietf.org/html/rfc5389
27. SocketCluster.io, http://socketcluster.io
28. Stream Control Transmission Protocol (SCTP), https://tools.ietf.org/html/rfc4960
29. Stoica I, Morris R, Karger D, Kaashoek MF, Balakrishnan H (2001) Chord: a scalable peer-to-peer lookup service for internet applications. In: Conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM). https://doi.org/10.1145/383059.383071
30. The WebSocket Protocol RFC6455, https://tools.ietf.org/html/rfc6455
31. The Secure Real-time Transport Protocol (SRTP), https://www.ietf.org/rfc/rfc3711.txt
32. Traversal Using Relays around NAT (TURN), https://tools.ietf.org/html/rfc5766

33. Vogt C, Werner MJ, Schmidt TC (2013) Leveraging WebRTC for P2P content distribution in web browsers. In: Proceedings of the International Conference on Network Protocols (ICNP). https://doi.org/10.1109/ICNP.2013.6733637
34. WebM, http://www.webmproject.org/
35. WebRTC, https://webrtc.org
36. WebRTC use cases, https://tools.ietf.org/html/rfc7478
37. WebSockets, https://www.websocket.org/
38. Wichtlhuber M, Rückert J, Stingl D, Schulz M, Hausheer D (2012) Energy-efficient mobile P2P video streaming. In: Proceedings of the IEEE 12th International Conference on Peer-to-Peer Computing (P2P). https://doi.org/10.1109/P2P.2012.6335812

Servio Palacios is a Fulbright Scholar and Ph.D. student with the Department of Computer Science at Purdue University. He holds an MSc in Computer Science from Purdue University, a Master in Information Security from Foro Europeo de Navarra, and a B.S. in Computer Systems from UNITEC, Honduras. Servio's research interests include graph theory, graph databases, distributed computing, edge computing, applied cryptography, and information security.

Victor Santos is a graduate of University Of Puerto Rico at Bayamon, where he received a bachelor?s degree in Computer Science and Purdue University, where he obtained a masters degree in Computer Science. He is currently a Computer Science doctoral student at Purdue University, and his research interests are graph theory, parallel computing, and programming languages.

Edgardo Barsallo is a PhD student in Computer Science at Purdue University, with particular interest in databases, distributed systems and mobile computing. He holds a bachelor?s degree in Computer Science from Universidad Tecnológica de Panamá.

Bharat Bhargava is a professor of the Department of Computer Science with a courtesy appointment in the School of Electrical and Computer Engineering at Purdue University. His research focuses on security and privacy issues in distributed systems. He is a Fellow of the Institute of Electrical and Electronics Engineers and of the Institute of Electronics and Telecommunication Engineers. In 1999, he received the IEEE Technical Achievement Award for his decade long contributions to foundations of adaptability in communication and distributed systems. He serves on seven editorial boards of international journals. He also serves the IEEE Computer Society on Technical Achievement Award and Fellow committees.